

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Exception Handling**

Inventor(s): .

**Scott D. Mosier**

**Ian H. Carmichael**

**Lawrence B. Sullivan**

**James J. Radigan**

**David N. Cutler**

ATTORNEY'S DOCKET NO. **MS1-1450US**

# Exception Handling

## TECHNICAL FIELD

This disclosure relates in general to exception handling and in particular, by way of example but not limitation, to facilitating exception handling with regard to functions of a runtime environment.

## BACKGROUND

A computer program typically includes many functions/methods that are executed while the computer program is running. The functions are executed by one or more processors in conjunction with at least one memory. The memory is used to store information for the functions, and such memory may include processor registers, cache memory, one or more stacks, main memory, some combination thereof, and so forth. A stack, for example, is usually employed to store information for multiple functions in a linear (e.g., temporal) manner.

FIG. 1 is a conventional stack 101 that illustrates an exemplary unwinding 105 thereof for handling an exception. As shown, stack 101 relates to multiple functions A, B, C ... L, M, and N. As each function is called or as one function transitions to another function, information for a function 103 is added to the stack. An example of such information is a frame or a call frame.

For example, information for function A 103(A) is at the bottom (or at least the lowest illustrated portion) of stack 101. When function B is called, information for function B 103(B) is added onto stack 101. Similarly, information for function C 103(C) ... information for function L 103(L), information for

1 function M 103(M), and information for function N 103(N) are gradually added to  
2 the stack.

3 Each of information for a function 103 may include such information as  
4 ongoing variable(s), stack pointer(s), instruction pointer(s), data in the registers of  
5 processor(s) that represents at least part of a current state of the processor(s), some  
6 combination thereof, and so forth. This information may be useful when an  
7 exception occurs:

8 Although modern programming entails significant debugging and testing,  
9 every imaginable event cannot be fully predicted. Such unexpected events can be  
10 accommodated and/or hidden from user view through exception handling  
11 routines/procedures. However, every function does not usually include error  
12 handling information. To reach a function that includes error handling  
13 information, stack 101 is unwound until information for a function 103 relates to a  
14 function that can handle unexpected events. In other words, information entries  
15 103 of stack 101 are traced back through, walked back up, etc. during a typical  
16 error handling procedure.

17 To that end, assuming function N cannot handle the experienced exception,  
18 stack unwinding 105NM is used to unwind stack 101 from a state appropriate for  
19 function N to a state appropriate for function M. If function M also does not  
20 possess appropriate error handling information, stack unwinding 105ML is used to  
21 unwind stack 101 from a state appropriate for function M to a state appropriate for  
22 function L. Stack 101 is thusly unwound until information for a function 103 that  
23 relates to a function that can handle the unexpected event is reached.

24 FIG. 2 is a conventional compiled file 201 in a static format. After a file is  
25 written by a programmer using an editor to produce source code, a compiler is

1 applied to the source code to produce object code in a machine language that is  
2 processor-consumable (e.g., a program executable file). The compiler is afforded  
3 the opportunity to consider all of the source code (possibly over multiple iterations  
4 of compiling) to “optimize” the organization of the object code while considering  
5 all objects, references, functions, error handling capabilities, and so forth. The  
6 compiler can thus produce a file 201 that is neatly organized in a predictable, static  
7 format.

8 An exemplary organization for file 201 includes first and second portions:  
9 code 203 and an unwind table or tables 205. Code 203 organizes individual code  
10 sections for functions A-N in an ordered fashion. Specifically, code 203 illustrates  
11 code for function A, code for function B, code for function C ... code for function  
12 L, code for function M, and code for function N.

13 Unwind table 205 is organized into three parts: unwind information 207,  
14 exception handling (EH) information 209, and code address (CA)-to-pointer  
15 information 211. Each of these three parts 207, 209, and 211 are subdivided into  
16 sections that are directed to particular functions. Specifically, each part 207, 209,  
17 and 211 illustrates sections for function N, for function M ... . Although not  
18 explicitly illustrated, each part 207, 209, and 211 may have sections for all  
19 functions A, B, C ... L, M, and N. In certain described implementations, unwind  
20 information 207 corresponds to so-called “r data”, exception handling information  
21 209 corresponds to so-called “x data”, and code address (CA)-to-pointer  
22 information 211 corresponds to so-called “p data”.

23 For CA-to-pointer information 211, each section that is directed to a  
24 particular respective function includes one or more of at least three entries: a start  
25 address, a final address, and an unwind pointer. The start address and the final

1 address relate to the addresses of the code for the respective function in code 203.  
2 These addresses may be relocatable virtual (RVA) addresses that are offsets from  
3 the beginning of code 203 and/or file 201. The unwind pointer is a reference that  
4 points to a section of unwind information 207 for the respective function. Thus,  
5 CA-to-pointer information 211 may imply that information from the address range  
6 of the coding to an unwind pointer for a function is provided, may imply that  
7 information for a mapping from instruction pointer (IP) addresses associated with  
8 a function to an unwind pointer thereof is provided, may imply that both  
9 information types are provided, and so forth.

10 As illustrated for unwind information 207, each section that is directed to a  
11 particular respective function includes at least two entries: an unwinding  
12 description and an exception handling (EH) pointer. The unwinding description  
13 describes how the stack can be unwound from the particular respective function  
14 back to the previous function. For example, an unwinding description of unwind  
15 information 207 for function N describes how to effectuate stack unwinding  
16 105NM for stack 101. Each exception handling pointer for a respective function is  
17 a reference that points to a section of exception handling information 209 for that  
18 respective function.

19 For exception handling information 209, each section (if present) includes  
20 exception handling information for a particular respective function. The exception  
21 handling information includes (native) exception handling tables or similar that  
22 explains how to handle one or more exceptions that have been experienced.

23 Illustrated file 201, and unwind table 205 thereof, can be effectively  
24 navigated quickly by an operating system (OS) when an exception occurs because  
25 it is cleanly and orderly organized. Consequently, standard computer science

1 algorithms targeted to searching for and/or locating desired information may be  
2 employed.

3 Unfortunately, code that is compiled on-the-fly and/or in ad hoc situations  
4 cannot be so easily organized logically and orderly in prescribed manners with  
5 predictable, static formats. Accordingly, there is a need for schemes and  
6 techniques that facilitate exception handling in a dynamic environment.

## 7 8 SUMMARY

9 In an exemplary media implementation, one or more electronically-  
10 accessible media include electronically-executable instructions that utilize an  
11 application programming interface, the application programming interface  
12 facilitating creation of callback-type dynamic function tables; each callback-type  
13 dynamic function table including a begin address, an end address, and a callback  
14 function, each callback-type dynamic function table corresponding to a code heap  
15 that stores code for multiple functions in a runtime environment; wherein  
16 interaction between the runtime environment and an operating system is  
17 precipitated upon calling the callback function to acquire exception handling  
18 and/or unwind information. In another exemplary media implementation, one or  
19 more electronically-accessible media include at least part of an operating system  
20 that is configured to request from a runtime environment exception handling  
21 and/or unwinding information for functions that are managed by the runtime  
22 environment.

23 In an exemplary electronic device implementation, an electronic device  
24 includes: a runtime environment that is managing code for multiple functions; and  
25 an operating system that is managing a linked list of dynamic function tables that

1 are searched when an exception occurs, the operating system adapted to call a  
2 callback function as indicated by a dynamic function table of the linked list of  
3 dynamic function tables to request that the runtime environment provide exception  
4 handling and/or unwind information for at least one function of the multiple  
5 functions; wherein the runtime environment is capable of providing to the  
6 operating system the exception handling and/or unwind information for the at least  
7 one function of the multiple functions responsive to the callback function.

8 In another exemplary media implementation, one or more electronically-  
9 accessible media include a data structure, the data structure including: a begin  
10 address; an end address; and a callback function that, when called, returns from a  
11 runtime environment exception handling and/or unwind information for a function  
12 associated with at least one address that is between the begin address and the end  
13 address.

14 In another exemplary media implementation, one or more electronically-  
15 accessible media include electronically-executable instructions that include: a  
16 callback function, the callback function accepting as input an instruction pointer  
17 that is associated with an address of a function from a runtime environment and  
18 producing as output data for code address-to-pointer information for the function  
19 having the address that is associated with the instruction pointer; wherein the  
20 callback function may be called by an operating system and implemented by the  
21 runtime environment.

22 Other method, system, approach, apparatus, application programming  
23 interface (API), device, media, procedure, arrangement, etc. implementations are  
24 described herein.  
25

## **BRIEF DESCRIPTION OF THE DRAWINGS**

The same numbers are used throughout the drawings to reference like and/or corresponding aspects, features, and components.

FIG. 1 is a conventional stack that illustrates an exemplary unwinding thereof for handling an exception.

FIG. 2 is a conventional compiled file in a static format.

FIG. 3 illustrates an exemplary dynamic function table (DFT) linked list.

FIG. 4 illustrates an exemplary approach to just-in-time (JIT) code compiling and a corresponding callback-type dynamic function table.

FIG. 5 illustrates an exemplary callback-type dynamic function table, which includes a callback function, and a related operating environment.

FIG. 6 illustrates an exemplary callback function and input/output parameters thereof.

FIG. 7 illustrates an exemplary code heap organization for JIT code compilation.

FIG. 8 is a flow diagram that illustrates an exemplary method for creating a callback-type dynamic function table.

FIG. 9 is a flow diagram that illustrates an exemplary method for using a callback-type dynamic function table.

FIG. 10 illustrates an exemplary computing (or general electronic device) operating environment that is capable of (wholly or partially) implementing at least one aspect of exception handling as described herein.



## **DETAILED DESCRIPTION**

FIG. 3 illustrates an exemplary dynamic function table (DFT) linked list 302. Dynamic function table linked list 302 includes multiple dynamic function tables 304. As illustrated, dynamic function table 304(1), dynamic function table 304(2) ... dynamic function table 304(n) are linked together into a list. Although not so illustrated, dynamic function table linked list 302 may alternatively be linked both backwards and forwards (e.g., doubly-linked) or linked in some other manner.

In a described implementation, dynamic function table linked lists 302 may be used, for example, for code that is not completely compiled into a static file. Each particular process may correspond to a particular dynamic function table 304. A header of the particular dynamic function table 304 includes a high level address range of the entire process, such as a first address and a last address of the process.

Generally, each dynamic function table 304 also includes a list of CA-to-pointer information for multiple functions. As shown for dynamic function table 304(1) specifically, a listing of “n” entries for CA-to-pointer information for functions #1, #2, #3 ... #n-1, and #n is included. Each CA-to-pointer information entry may include, for example, a start address, a final address, and an unwind pointer for the respective function to which the entry is directed. In certain described implementations, each entry of CA-to-pointer information for a given function may correspond to a so-called “runtime function entry”.

Hence, each dynamic function table 304 is directed to a process and includes multiple CA-to-pointer information entries. Each respective CA-to-pointer information entry is directed to a respective function of the process and

1 includes CA-to-pointer information for the respective function. The listing of CA-  
2 to-pointer information entries in any given dynamic function table 304 may be  
3 sorted or unsorted.

4 When an exception is experienced, an OS of the device or system uses the  
5 relevant current instruction pointer to check a high level range of each header for  
6 each dynamic function table 304 until a match is found. In other words, the OS  
7 moves along dynamic function table linked list 302 from one dynamic function  
8 table 304 to the next and checks to determine if the instruction pointer is an  
9 address that is between the first address and the last address. If so, a dynamic  
10 function table 304 match is found, and a CA-to-pointer information entry for the  
11 particular function at issue is (assuming no errors) present somewhere in that  
12 matching dynamic function table 304.

13 The OS then traverses the listing of CA-to-pointer information entries in  
14 the matched dynamic function table 304 until the CA-to-pointer information entry  
15 that has a start address and a final address defining a range that includes the  
16 instruction pointer is located. This CA-to-pointer information entry is for the  
17 particular function at issue. The OS can then use the unwind pointer in this CA-  
18 to-pointer information entry to locate unwind information and proceed to handling  
19 the exception.

20 Thus, when an exception occurs, the OS moves along dynamic function  
21 table linked list 302 until a matching dynamic function table 304 is found, and it  
22 then traverses the entries thereof until the matching CA-to-pointer information  
23 entry for the function currently being examined or unwound is located. Due to the  
24 linear nature of this approach, it is satisfactory as long as the length of dynamic  
25 function table linked list 302 does not become too excessive, especially to the

1 extent that sorted dynamic function tables 304 are utilized. However, in a runtime  
2 environment, the length of dynamic function table linked list 302 can quickly  
3 become unwieldy and unmanageable.

4 FIG. 4 illustrates an exemplary approach 400 to just-in-time (JIT) code  
5 compiling 404 and a corresponding callback-type dynamic function table 410. JIT  
6 compiling 404 operates in runtime environment (RTE) 406 to create machine  
7 language code as it is requested. This JIT code creation may be performed at a  
8 function-by-function granularity. An example of a runtime environment 406 is the  
9 common language runtime (CLR) of Microsoft® Corporation.

10 In a described implementation with JIT compiling 404, intermediate code  
11 402 is provided that includes one or more functions. Optionally, intermediate code  
12 402 may be platform independent (e.g., “write-once”) code. JIT compiling 404  
13 transforms intermediate code 402 into compiled code 408 that includes one or  
14 more functions that may be consumed by a processor. However, instead of  
15 compiling the entirety of intermediate code 402 to produce one static file for  
16 compiled code 408, JIT compiling 404 compiles parts of intermediate code 402  
17 into portions of compiled code 408. This JIT compiling 404 may be performed as  
18 it is needed, as it is requested, as time permits, and so forth.

19 As illustrated, intermediate code 402 includes four functions: function  
20 One( ), function Two( ), function Three( ), and function Four( ). JIT compiler 404  
21 compiles these four respective intermediate code 402 parts into four respective  
22 portions of compiled code 408: function One( ), function Two( ), function Three(  
23 ), and function Four( ). However, the order of the functions has changed during  
24 the compiling because the compiling is performed as each function is called,  
25

1 which may not be in the same order as the functions are presented within  
2 intermediate code 402.

3 Thus, functions may be compiled in an unknown, unpredictable, and/or  
4 non-sequential order by JIT compiler 404 in a runtime environment 406.  
5 Furthermore, one or more functions of intermediate code 402 may not be compiled  
6 into one or more functions for compiled code 408 in any given session for runtime  
7 environment 406. For example, a given function of intermediate code 402 may  
8 not be compiled for compiled code 408 until it is called, and it may not be called  
9 in a given session.

10 When executing functions of compiled code 408, exceptions can be  
11 experienced. Consequently, an exception handling mechanism is also employed  
12 for compiled code 408. One approach is to create at least one dynamic function  
13 table 304 for compiled code 408. However, each dynamic function table 304 is  
14 established based on a high level range of addresses, and such a high level range  
15 of addresses for multiple functions is not known for compiled code 408 because  
16 functions are being dynamically added thereto.

17 Another approach is to create a new dynamic function table 304 for each  
18 newly compiled function of compiled code 408. The first address and the last  
19 address, as well as an unwind pointer (e.g., for the CA-to-pointer information), is  
20 known and/or can be determined by runtime environment 406 at the time of  
21 compiling for any given individual function. However, in a runtime environment  
22 406 with JIT compiling 404, there may be thousands and thousands of such JIT  
23 compiled functions for compiled code 408. Dynamic function table linked list 302  
24 can therefore extend to over 100,000 dynamic function tables 304 that are linked  
25

1 together. Searching such a linked list is time consuming and can degrade  
2 performance.

3 Yet another approach is to further rely on runtime environment 406 to  
4 manage some of the exception handling responsibilities for compiled code 408  
5 that has been JIT compiled 404. This approach is described further herein. For  
6 example, a callback dynamic function table 410 that corresponds to multiple  
7 functions of compiled code 408 can be employed. This correspondence 414  
8 between multiple functions of compiled code 408 and callback dynamic function  
9 table 410 is described further below.

10 Callback dynamic function table 410 includes a callback function 412.  
11 Callback dynamic function table 410 may be used in a separate dynamic function  
12 table linked list 302 with no dynamic function tables 304, or callback dynamic  
13 function table 410 may be included in a dynamic function table linked list 302  
14 with sorted and/or unsorted dynamic function tables 304.

15 When handling an exception for a given function of compiled code 408 that  
16 corresponds to callback dynamic function table 410, the OS that is performing the  
17 exception dispatching initiates callback function 412. Initiating callback function  
18 412 notifies runtime 406 of an exception with respect to the given function.  
19 Responsive to a call to callback function 412, runtime 406 consults the code, state,  
20 and/or other information that it is managing, especially as they relate to the given  
21 function. Runtime 406 responds to the OS with information that is sufficient to  
22 enable the OS to continue with exception handling. For example, runtime 406 can  
23 provide the OS with exception handling and/or unwind information by (i)  
24 determining and (ii) passing a reference to CA-to-pointer information for the given  
25 function to the OS.

1        FIG. 5 illustrates an exemplary callback-type dynamic function table 410,  
2        which includes a callback function 412, and a related operating environment. The  
3        operating environment includes operating system (OS) 508 and runtime  
4        environment 406 (or runtime 406/runtime component 406).

5        In a described implementation, OS 508 manages callback-type dynamic  
6        function tables 410 that include a callback function 412. This management may  
7        be separate from the management of any other exception handling/unwinding  
8        mechanisms. Alternatively, OS 508 may manage callback-type dynamic function  
9        tables 410 in conjunction with the management of dynamic function tables 304 of  
10       dynamic function table linked list 302. For example, callback-type dynamic  
11       function tables 410 may be linked into dynamic function table linked list 302  
12       along with dynamic function tables 304.

13       As illustrated, runtime environment 406 includes compiled code 408.  
14       Compiled code 408 includes compiled versions of function One( ), function Two(  
15       ), function Three( ), function Four( ), and so forth. As noted above, with JIT  
16       compiling 404, functions for a given process are compiled from intermediate code  
17       402 when (and if) requested. Consequently, the size that is ultimately occupied by  
18       the given process is often unknown as a first function, such as function One( ) of  
19       compiled code 408, is being executed by a processor.

20       To account for this unknown ultimate size for the given process, runtime  
21       environment 406 allocates (or has allocated) a chunk of memory of a relatively  
22       arbitrary size from one or more heaps. This chunk of memory may then be shared  
23       by multiple processes as well as by multiple functions.

24       Code heap 502 is a chunk of memory that has been allocated for runtime  
25       environment 406 by runtime environment 406 and/or by OS 508. Code heap 502

1 has a begin address 504 and an end address 506. Although not so explicitly  
2 illustrated in FIG. 5, code heap 502 may include functions and other information  
3 for multiple processes, as well as heap managing information for use by runtime  
4 406.

5 In a described implementation, there is a correspondence between code  
6 heap 502 and callback dynamic function table 410. Callback dynamic function  
7 table 410 includes (e.g., values for) begin address 504 and end address 506 of code  
8 heap 502. In other words, callback dynamic function table 410 corresponds to  
9 code for functions between begin address 504 and end address 506 of code heap  
10 502, and begin address 504 and end address 506 of callback dynamic function  
11 table 410 reflects this correspondence.

12 As a result, each callback-type dynamic function table 410 may correspond  
13 to multiple functions for one process and/or multiple functions across multiple  
14 processes. Because each callback-type dynamic function table 410 corresponds to  
15 multiple functions, dynamic function table linked list 302 is grown at a  
16 significantly reduced rate (as compared to one dynamic function table 304 per  
17 function). Dynamic function table linked list 302 is therefore shorter and more  
18 quickly searched and otherwise more easily managed by OS 508.

19 In an example use of callback dynamic function table 410, when an  
20 exception is discovered and exception handling/unwind information is being  
21 acquired by OS 508 for a current function associated with a current instruction  
22 pointer, OS 508 searches dynamic function table linked list 302. OS 508 searches  
23 dynamic function table linked list 302 by moving along dynamic function tables  
24 304 and/or callback-type dynamic function tables 410 until address checking  
25 determines that the current instruction pointer is between begin address 504 and

1 end address 506 of callback dynamic function table 410. OS 508 then initiates  
2 callback function 412 of callback dynamic function table 410. When OS 508  
3 makes a call to callback function 412, OS 508 is effectively asking/requesting  
4 runtime environment 406 for help in acquiring exception handling/unwind  
5 information. Responsive to callback function 412, runtime 406 provides  
6 information to OS 508 to help with exception handling/unwinding.

7 FIG. 6 illustrates an exemplary callback function 412 and input/output  
8 parameters thereof. Exemplary input parameters for callback function 412 are an  
9 instruction pointer and a callback dynamic function table reference for context.  
10 An exemplary output parameter for callback function 412 is a reference to CA-to-  
11 pointer information.

12 When OS 508 calls callback function 412, OS 508 includes a current  
13 instruction pointer and a reference to the callback dynamic function table 410 that  
14 was discovered to have a begin address 504 and an end address 506 that jointly  
15 form a range that contains the current instruction pointer. Thus, the current  
16 instruction pointer and the callback dynamic function table reference are passed  
17 from OS 508 to runtime environment 406 for callback function 412. Runtime 406  
18 performs the exception handling/unwinding callback analysis for callback function  
19 412 and passes back to OS 508 a reference to CA-to-pointer information for the  
20 function associated with the current instruction pointer. This analysis is described  
21 further below with particular reference to FIGS. 7 and 9.

22 The reference to CA-to-pointer information references CA-to-pointer  
23 information for the function currently at issue (e.g., being unwound and/or  
24 considered for exception handling abilities). This reference to CA-to-pointer  
25 information may be, for example, a pointer to CA-to-pointer information that is



1 stored in a code heap 502 by runtime 406 for the function at issue. Upon  
2 following the reference, OS 508 may attain the CA-to-pointer information for the  
3 function of the current instruction. This CA-to-pointer information may include a  
4 start address, a final address, and an unwind pointer for the function that is  
5 associated with the current instruction. This information as it relates to the  
6 associated function at issue is described further below with reference to FIG. 7.

7 FIG. 7 illustrates an exemplary code heap organization 700 for JIT code  
8 compilation 404. A runtime 406 (of FIGS. 4-6) establishes a chunk of code in  
9 which runtime 406 is to store code for functions, heap managing information, and  
10 so forth. This chunk of code, or code heap 502, has a begin address 504 and an  
11 end address 506. Begin address 504 and end address 506 of code heap 502 are  
12 known by and/or provided to an OS 508 (of FIGS. 5 and 6). Although one code  
13 heap 502 is illustrated, each runtime environment 406 may have two or more such  
14 code heaps 502.

15 As illustrated, code heap 502 includes a heap structure 702, a code for  
16 function One 704, code for (one or more) other functions 718, and so forth. Heap  
17 structure 702 includes information for managing (e.g., organizing, controlling,  
18 etc.) code heap 502. For example, heap structure 702 may include contents (e.g., a  
19 table, an index, etc.) that identifies functions that are included as part of code heap  
20 502. These identifications may include the address locations/ranges of different  
21 code segments for different functions of code heap 502, such as code for function  
22 One 704.

23 Code for function One 704 is bounded by start address 706 and final  
24 address 708. Code for function One 704 includes multiple portions: a code  
25 header 710, function One 712, unwind information 714, and CA-to-pointer

1 information 716. Each of these portions of code for function One 704, as  
2 indicated by the dashed line in code header 710, may include aspects in addition to  
3 those illustrated in FIG. 7 and described below.

4 Function One 712 includes the machine language, chip-consumable  
5 instructions for function One. Code header 710 includes a reference to CA-to-  
6 pointer information, which points to CA-to-pointer information 716.

7 CA-to-pointer information 716 includes a start address, a final address, and  
8 an unwind pointer. The values of the start address and the final address equate to  
9 and/or reflect start address 706 and final address 708, respectively. The unwind  
10 pointer references unwind information 714.

11 Unwind information 714 includes unwind information and an exception  
12 handling pointer. The exception handling pointer, if present, references exception  
13 handling information for function One. The unwinding description describes how  
14 to unwind the stack from function One to the preceding function. This unwinding  
15 description may be used by, for example, OS 508 to unwind the stack.

16 Each of these addresses, such as start address 706 and final address 708,  
17 may be RVA addresses that are offsets from or relative to begin address 504 of  
18 code heap 502. Hence, the start address and the final address of CA-to-pointer  
19 information 716 may be stored as RVA addresses. Similarly, both of (i) the  
20 reference to CA-to-pointer information of code header 710 and (ii) the unwind  
21 pointer of CA-to-pointer information 716 may effectuate their respective  
22 references relative to begin address 504.

23 Application of and interaction with code heap organization 700 in the  
24 context of using and creating a callback dynamic function table 410 is described  
25 further below with reference to FIG. 8 and FIG. 9, respectively.

1        FIG. 8 is a flow diagram 800 that illustrates an exemplary method for  
2        creating a callback-type dynamic function table. Flow diagram 800 includes four  
3        (4) blocks 802-808. Although the actions of flow diagram 800 may be performed  
4        in any operating environment, FIGS. 3-7 are used to illuminate certain aspects of  
5        the method.

6        At block 802, a new code heap is initialized with a runtime environment.  
7        For example, runtime 406 may initialize code heap 502. At block 804, a begin  
8        address and an end address of the new code heap is noted. For example, runtime  
9        406 may note begin address 504 and end address 506 for code heap 502. It should  
10       be noted that the new code heap initialization may be effectuated with the  
11       cooperation of an allocator or similar component that may be a constituent of OS  
12       508.

13       At block 806, a dynamic function table is created that corresponds to the  
14       code heap. The dynamic function table is created with the noted begin address,  
15       the noted end address, and a callback function. For example, a callback dynamic  
16       function table 410 that includes begin address 504, end address 506, and callback  
17       function 412 may be created by runtime 406, by OS 508, by a combination of  
18       runtime 406 and OS 508, and so forth.

19       In a described implementation, runtime 406 causes OS 508 to create  
20       callback dynamic function table 410 by requesting an addition to dynamic  
21       function table linked list 302. For example, runtime 406 can use an application  
22       programming interface (API) to request that OS 508 create a new table in dynamic  
23       function table linked list 302.

24       By way of example only, for a Microsoft® Windows® OS, a runtime  
25       component can call an Install Dynamic Function Table(table, first address, last

1 address, ... ) API, with the table parameter set to callback dynamic function table,  
2 the first address set to begin address, and the last address set to end address. The  
3 table parameter may also be set to sorted dynamic function table or unsorted  
4 dynamic function table when installing non-callback-type dynamic function tables.  
5 Generally, OS 508 may also provide an API for use by runtime 406 in order to  
6 remove a dynamic function table (including callback-type dynamic function tables  
7 410) from dynamic function table linked list 302.

8 At block 808, a callback-type dynamic function table for the code heap is  
9 added to a linked list of dynamic function tables. For example, callback dynamic  
10 function table 410, which includes begin address 504, end address 506, and  
11 callback function 412, may be added to dynamic function table linked list 302. If  
12 dynamic function table linked list 302 is managed by OS 508, then OS 508 adds  
13 callback dynamic function table 410 thereto.

14 FIG. 9 is a flow diagram 900 that illustrates an exemplary method for using  
15 a callback-type dynamic function table. Flow diagram 900 includes ten (10)  
16 blocks 902-920. Although the actions of flow diagram 900 may be performed in  
17 any operating environment, FIGS. 3-7 are used to illuminate certain aspects of the  
18 method. Moreover, flow diagram 900 is divided into two parts: OS 508 and  
19 runtime environment 406. As illustrated, OS 508 performs the actions of blocks  
20 902-908 and blocks 918-920, and runtime environment 406 performs the actions  
21 of blocks 910-916.

22 At block 902, an exception is discovered that results from execution of a  
23 function in a runtime environment. For example, an exception may result from  
24 execution of function One (using the machine language of function One 712) in  
25 runtime environment 406. It should be noted that OS 508 may not be aware at the

1 time of the exception that the code responsible for the exception is being executed  
2 as part of runtime environment 406.

3 With the discovery of the exception, OS 508 begins an exception handling  
4 procedure that typically includes unwinding one or more frames of the stack.  
5 Although the description herein focuses on exception handling as a result of an  
6 exception that occurs with/in runtime environment 406, exceptions also occur at  
7 addresses that fall within the range of currently loaded static executable images.  
8 OS 508 therefore considers, at least from time to time, both static unwind tables  
9 (not explicitly shown in FIGS. 3-10) for files that are not part of runtime  
10 environment 406 and at least one dynamic function table linked list 302.  
11 Generally, upon discovery of an exception, OS 508 first searches the static unwind  
12 tables. If the corresponding unwinding information, etc. cannot be attained  
13 through the static unwind tables, OS 508 then searches dynamic function table  
14 linked list 302. However, OS 508 may alternatively search dynamic function table  
15 linked list 302 prior to and/or overlapping with a search of the static unwind  
16 tables.

17 At block 904, a dynamic function table linked list is searched using the  
18 current instruction pointer. For example, the high level ranges of sorted/unsorted  
19 dynamic function tables 304 may be searched as well as begin/end addresses  
20 504/506 of callback-type dynamic function tables 410 as OS 508 moves along  
21 dynamic function tables of dynamic function table linked list 302.

22 At block 906, the callback dynamic function table for the current  
23 instruction pointer is located. For example, a callback dynamic function table 410  
24 that has a begin address 504 that is lower than and an end address 506 that is  
25 higher than the current instruction pointer may be located.

1       At block 908, the callback function is initiated. For example, OS 508 can  
2 call callback function 412 using the current instruction pointer and a reference to  
3 the located callback dynamic function table 410 as arguments for the call.  
4 Initiating the callback function serves to notify runtime environment 406 that OS  
5 508 is requesting exception handling and/or unwinding information to handle an  
6 exception with regard to the current instruction pointer.

7       At block 910, responsive to initiation of the callback function and the  
8 current instruction pointer, code for the runtime function is ascertained. For  
9 example, runtime environment 406 may consult a global table that covers multiple  
10 code heaps 502. Such a global table includes an entry for each runtime function  
11 that maps (i) start address 706/final address 708 code ranges for each code for  
12 functions 704/718 to (ii) code headers 710. If a global table is employed, the  
13 mapping may also include an identification of the relevant code heap 502,  
14 especially if RVA addresses are used.

15       Alternatively, a table (of multiple tables) with mappings similar to those  
16 described above for a global table may be included at each heap structure 702  
17 where the table is for the functions included in the associated code heap 502. If  
18 multiple such per-code-heap tables are employed, runtime environment 406  
19 searches each table at each heap structure 702 until the function associated with  
20 the current instruction pointer is ascertained. Runtime environment 406 may also  
21 maintain a table that maps code heaps 502 to address ranges defined by begin/end  
22 addresses 504/506 so that the relevant heap structure 702 may be ascertained  
23 without searching through multiple heap structures 702.

24       At block 912, a code header of the runtime function is accessed. For  
25 example, once the code for the function associated with the current instruction

1 pointer is ascertained, then code header 710 therefor may be accessed. In short,  
2 the actions of blocks 910 and 912 may correspond to (i) finding a code heap 502  
3 for the current instruction pointer and (ii) finding/accessing a code header 710 of  
4 code heap 502 for the current instruction pointer. In the description that follows,  
5 code for function One 704 is used as an example. Thus, code header 710 of code  
6 for function One 704 may be inspected as part of the accessing.

7 At block 914, a reference to CA-to-pointer information is extracted from  
8 the code header of the runtime function. For example, the reference to CA-to-  
9 pointer information may be extracted from code header 710. At block 916, the  
10 reference to CA-to-pointer information is passed to the operating system as a  
11 response to and/or output of the callback function. For example, the reference to  
12 CA-to-pointer information of code header 710 that references CA-to-pointer  
13 information 716 may be provided from runtime environment 406 to OS 508.

14 More generally, runtime environment 406 may provide to OS 508 data for  
15 CA-to-pointer information. This data for CA-to-pointer information may  
16 comprise a reference to CA-to-pointer information 716. Alternatively, this data for  
17 CA-to-pointer information may comprise CA-to-pointer information 716. In other  
18 words, runtime environment 406 may alternatively directly provide to OS 508 the  
19 start address value, the final address value, and/or an unwind pointer for code for  
20 function One 704.

21 At block 918, the reference to CA-to-pointer information is used to attain  
22 the CA-to-pointer information. This CA-to-pointer information includes an  
23 unwind pointer that points to unwind information. For example, OS 508 may use  
24 the reference to CA-to-pointer information, possibly in conjunction with begin  
25 address 504 if relative addressing is employed, to attain the CA-to-pointer

1 information from CA-to-pointer information 716. CA-to-pointer information 716  
2 includes an unwind pointer that points to unwind information 714.

3 At block 920, the unwind pointer is used to attain unwind information. For  
4 example, the unwind pointer attained from CA-to-pointer information 716 may be  
5 used by OS 508 to access unwind information 714 and to extract the unwinding  
6 description therefrom. OS 508 may then unwind the frame on the stack that is  
7 associated with function One.

8 The aspects, features, components, etc. of FIGS. 3-7 and the methods of  
9 FIGS. 8 and 9, for example, are illustrated in diagrams that are divided into  
10 multiple blocks. However, the order and/or layout in which the operating  
11 environments and methods are described and/or shown is not intended to be  
12 construed as a limitation, and any number of the blocks can be combined,  
13 rearranged, augmented, omitted, etc. in any manner to implement one or more  
14 systems, methods, devices, procedures, media, APIs, apparatuses, arrangements,  
15 etc. for exception handling. Furthermore, although the description herein includes  
16 references to specific implementations such as those of FIGS. 3-7 (as well as the  
17 exemplary operating environment of FIG. 10), the operating environments and  
18 methods can be implemented in any suitable hardware, software, firmware, or  
19 combination thereof and using any suitable runtime language(s), runtime  
20 environment(s), application programming interface(s), memory structure(s), and  
21 so forth.

22 FIG. 10 illustrates an exemplary computing (or general electronic device)  
23 operating environment 1000 that is capable of (fully or partially) implementing at  
24 least one system, device, apparatus, component, arrangement, protocol, approach,  
25 method, procedure, API, some combination thereof, etc. for exception handling as



1 described herein. Computing environment 1000 may be utilized in the computer  
2 and network architectures described below or in a stand-alone situation.

3 Exemplary electronic device operating environment 1000 is only one  
4 example of an environment and is not intended to suggest any limitation as to the  
5 scope of use or functionality of the applicable electronic (including computer,  
6 game console, television, etc.) architectures. Neither should electronic device  
7 environment 1000 be interpreted as having any dependency or requirement  
8 relating to any one or to any combination of components as illustrated in FIG. 10.

9 Additionally, exception handling may be implemented with numerous other  
10 general purpose or special purpose electronic device (including computing system)  
11 environments or configurations. Examples of well known electronic (device)  
12 systems, environments, and/or configurations that may be suitable for use include,  
13 but are not limited to, personal computers, server computers, thin clients, thick  
14 clients, personal digital assistants (PDAs) or mobile telephones, watches, hand-  
15 held or laptop devices, multiprocessor systems, microprocessor-based systems,  
16 set-top boxes, programmable consumer electronics, video game machines, game  
17 consoles, portable or handheld gaming units, network PCs, minicomputers,  
18 mainframe computers, distributed or multi-processing computing environments  
19 that include any of the above systems or devices, some combination thereof, and  
20 so forth.

21 Implementations for exception handling may be described in the general  
22 context of electronically-executable instructions. Generally, electronically-  
23 executable instructions include routines, programs, objects, components, data  
24 structures, etc. that perform particular tasks or implement particular abstract data  
25 types. Exception handling, as described in certain implementations herein, may

1 also be practiced in distributed computing environments where tasks are  
2 performed by remotely-linked processing devices that are connected through a  
3 communications link and/or network. Especially in a distributed computing  
4 environment, electronically-executable instructions may be located in separate  
5 storage media, executed by different processors, and/or propagated over  
6 transmission media.

7       Electronic device environment 1000 includes a general-purpose computing  
8 device in the form of a computer 1002, which may comprise any electronic device  
9 with computing and/or processing capabilities. The components of computer 1002  
10 may include, but are not limited to, one or more processors or processing units  
11 1004, a system memory 1006, and a system bus 1008 that couples various system  
12 components including processor 1004 to system memory 1006.

13       System bus 1008 represents one or more of any of many types of wired or  
14 wireless bus structures, including a memory bus or memory controller, a point-to-  
15 point connection, a switching fabric, a peripheral bus, an accelerated graphics port,  
16 and a processor or local bus using any of a variety of bus architectures. By way of  
17 example, such architectures may include an Industry Standard Architecture (ISA)  
18 bus, a Micro Channel Architecture (MCA) bus, an Enhanced ISA (EISA) bus, a  
19 Video Electronics Standards Association (VESA) local bus, a Peripheral  
20 Component Interconnects (PCI) bus also known as a Mezzanine bus, some  
21 combination thereof, and so forth.

22       Computer 1002 typically includes a variety of electronically-accessible  
23 media. Such media may be any available media that is accessible by computer  
24 1002 or another electronic device, and it includes both volatile and non-volatile  
25 media, removable and non-removable media, and storage and transmission media.

1        System memory 1006 includes electronically-accessible storage media in  
2 the form of volatile memory, such as random access memory (RAM) 1010, and/or  
3 non-volatile memory, such as read only memory (ROM) 1012. A basic  
4 input/output system (BIOS) 1014, containing the basic routines that help to  
5 transfer information between elements within computer 1002, such as during start-  
6 up, is typically stored in ROM 1012. RAM 1010 typically contains data and/or  
7 program modules/instructions that are immediately accessible to and/or being  
8 presently operated on by processing unit 1004.

9        Computer 1002 may also include other removable/non-removable and/or  
10 volatile/non-volatile storage media. By way of example, FIG. 10 illustrates a hard  
11 disk drive or disk drive array 1016 for reading from and writing to a (typically)  
12 non-removable, non-volatile magnetic media (not separately shown); a magnetic  
13 disk drive 1018 for reading from and writing to a (typically) removable, non-  
14 volatile magnetic disk 1020 (e.g., a "floppy disk"); and an optical disk drive 1022  
15 for reading from and/or writing to a (typically) removable, non-volatile optical  
16 disk 1024 such as a CD-ROM, DVD, or other optical media. Hard disk drive  
17 1016, magnetic disk drive 1018, and optical disk drive 1022 are each connected to  
18 system bus 1008 by one or more storage media interfaces 1026. Alternatively,  
19 hard disk drive 1016, magnetic disk drive 1018, and optical disk drive 1022 may  
20 be connected to system bus 1008 by one or more other separate or combined  
21 interfaces (not shown).

22        The disk drives and their associated electronically-accessible media provide  
23 non-volatile storage of electronically-executable instructions, such as data  
24 structures, program modules, and other data for computer 1002. Although  
25 exemplary computer 1002 illustrates a hard disk 1016, a removable magnetic disk

1 1020, and a removable optical disk 1024, it is to be appreciated that other types of  
2 electronically-accessible media may store instructions that are accessible by an  
3 electronic device, such as magnetic cassettes or other magnetic storage devices,  
4 flash memory, CD-ROM, digital versatile disks (DVD) or other optical storage,  
5 RAM, ROM, electrically-erasable programmable read-only memories (EEPROM),  
6 and so forth. Such media may also include so-called special purpose or hard-  
7 wired integrated circuit (IC) chips. In other words, any electronically-accessible  
8 media may be utilized to realize the storage media of the exemplary electronic  
9 system and environment 1000.

10 Any number of program modules (or other units or sets of  
11 instructions/code) may be stored on hard disk 1016, magnetic disk 1020, optical  
12 disk 1024, ROM 1012, and/or RAM 1010, including by way of general example,  
13 an operating system 1028, one or more application programs 1030, other program  
14 modules 1032, and program data 1034. By way of example but not limitation,  
15 operating system 1028 may correspond to OS 508.

16 A user may enter commands and/or information into computer 1002 via  
17 input devices such as a keyboard 1036 and a pointing device 1038 (e.g., a  
18 "mouse"). Other input devices 1040 (not shown specifically) may include a  
19 microphone, joystick, game pad, satellite dish, serial port, scanner, and/or the like.  
20 These and other input devices are connected to processing unit 1004 via  
21 input/output interfaces 1042 that are coupled to system bus 1008. However, input  
22 devices and/or output devices may instead be connected by other interface and bus  
23 structures, such as a parallel port, a game port, a universal serial bus (USB) port,  
24 an infrared port, an IEEE 1394 ("Firewire") interface, an IEEE 802.11 wireless  
25 interface, a Bluetooth® wireless interface, and so forth.

1 A monitor/view screen 1044 or other type of display device may also be  
2 connected to system bus 1008 via an interface, such as a video adapter 1046.  
3 Video adapter 1046 (or another component) may be or may include a graphics  
4 card for processing graphics-intensive calculations and for handling demanding  
5 display requirements. Typically, a graphics card includes a graphics processing  
6 unit (GPU), video RAM (VRAM), etc. to facilitate the expeditious performance of  
7 graphics operations. In addition to monitor 1044, other output peripheral devices  
8 may include components such as speakers (not shown) and a printer 1048, which  
9 may be connected to computer 1002 via input/output interfaces 1042.

10 Computer 1002 may operate in a networked environment using logical  
11 connections to one or more remote computers, such as a remote computing device  
12 1050. By way of example, remote computing device 1050 may be a personal  
13 computer, a portable computer (e.g., laptop computer, tablet computer, PDA,  
14 mobile station, etc.), a palm or pocket-sized computer, a watch, a gaming device, a  
15 server, a router, a network computer, a peer device, other common network node,  
16 or another electronic device type as listed above, and so forth. However, remote  
17 computing device 1050 is illustrated as a portable computer that may include  
18 many or all of the elements and features described herein with respect to computer  
19 1002.

20 Logical connections between computer 1002 and remote computer 1050 are  
21 depicted as a local area network (LAN) 1052 and a general wide area network  
22 (WAN) 1054. Such networking environments are commonplace in offices,  
23 enterprise-wide computer networks, intranets, the Internet, fixed and mobile  
24 telephone networks, ad-hoc and infrastructure wireless networks, other wireless  
25 networks, gaming networks, some combination thereof, and so forth.

1       When implemented in a LAN networking environment, computer 1002 is  
2 usually connected to LAN 1052 via a network interface or adapter 1056. When  
3 implemented in a WAN networking environment, computer 1002 typically  
4 includes a modem 1058 or other means for establishing communications over  
5 WAN 1054. Modem 1058, which may be internal or external to computer 1002,  
6 may be connected to system bus 1008 via input/output interfaces 1042 or any  
7 other appropriate mechanism(s). It is to be appreciated that the illustrated network  
8 connections are exemplary and that other means of establishing communication  
9 link(s) between computers 1002 and 1050 may be employed.

10       In a networked environment, such as that illustrated with electronic device  
11 environment 1000, program modules or other instructions that are depicted  
12 relative to computer 1002, or portions thereof, may be fully or partially stored in a  
13 remote memory storage device. By way of example, remote application programs  
14 1060 reside on a memory component of remote computer 1050 but may be usable  
15 or otherwise accessible via computer 1002. Also, for purposes of illustration,  
16 application programs 1030 and other electronically-executable instructions such as  
17 operating system 1028 are illustrated herein as discrete blocks, but it is recognized  
18 that such programs, components, and other instructions reside at various times in  
19 different storage components of computing device 1002 (and/or remote computing  
20 device 1050) and are executed by data processor(s) 1004 of computer 1002 (and/or  
21 those of remote computing device 1050).

22       Although systems, media, devices, methods, procedures, apparatuses,  
23 techniques, approaches, procedures, arrangements, and other implementations  
24 have been described in language specific to structural, logical, algorithmic, and  
25 functional features and/or diagrams, it is to be understood that the invention

1 defined in the appended claims is not necessarily limited to the specific features or  
2 diagrams described. Rather, the specific features and diagrams are disclosed as  
3 exemplary forms of implementing the claimed invention.